CSE 130: Fall 2012
Programming Languages

Lecture 10: Polymorphism

Ranjit Jhala
UC San Diego

Thursday, November 1, 2012
Example: Calculator Revisited

```ocaml
type expr =  
|  Num of int  
|  Div of expr * expr
```

Can you write a function?

```ocaml
val eval : expr -> int
```
In Class Exercise

type expr =
  | Num of int
  | Div of expr * expr

Write an Evaluation function

val eval : expr -> int option

That returns None if a div-by-zero occurs
Moral

Failure *is* an Option!
Datatypes with many type variables

type ('a, 'b) tree =
  Leaf
| Node of 'a * 'b * ('a, 'b) tree * ('a, 'b) tree
Datatypes with many type variables

• Multiple type variables

```ocaml
type ('a,'b) tree = 
  Leaf 
| Node of 'a* 'b * ('a,'b) tree * ('a,'b) tree
```

• Type is instantiated for each use:

```ocaml
Node("alice", 2, Leaf, Leaf)
Node("charlie", 3, Leaf, Leaf)
Node("bob", 13,
  , Node("alice", 2, Leaf, Leaf)
  , Node("charlie", 3, Leaf, Leaf))
```
Datatypes with many type variables

- Multiple type variables

```haskell
type ('a,'b) tree = 
  Leaf
| Node of 'a* 'b * ('a,'b) tree * ('a,'b) tree
```

- Type is instantiated for each use:

```
Node(“alice”, 2, Leaf, Leaf)
Node(“charlie”, 3, Leaf, Leaf)
Node(“bob”, 13,
    , Node(“alice”, 2, Leaf, Leaf)
    , Node(3, “charlie”, Leaf, Leaf))
```
Binary Search Trees

\[
\text{type ('a, 'b) tree =}
\begin{align*}
\text{Leaf} \\
| \text{Node of 'a* 'b* ('a,'b) tree * ('a,'b) tree}
\end{align*}
\]

Node (key, value, left, right)

BST Property:
keys in left < key < keys in right
BST Property: *keys in left < key < keys in right*

```
Node(“bob”, 13,
    Node(“alice”, 2, Leaf, Leaf),
    Node(“charlie”, 3, Leaf, Leaf))
```
In-Class Exercise!

BST Property: \textit{keys in left < key < keys in right}

\begin{verbatim}
type (\texttt{a}, \texttt{b}) tree =
   Leaf
| Node of \texttt{a} * \texttt{b} * ((\texttt{a},\texttt{b}) tree * ((\texttt{a},\texttt{b}) tree
\end{verbatim}

Write a function to lookup keys...

\begin{verbatim}
val \texttt{lookup}: \texttt{a} -\rightarrow ((\texttt{a},\texttt{b}) tree -\rightarrow \texttt{b} option
\end{verbatim}
Polymorphic Data Structures

- **Container** data structures independent of type!
- Appropriate type is *instantiated* at each *use*:
  
  ```
  'a list
  ('a, 'b) tree
  ('a, 'b) hashtbl ...
  ```

- **Static type checking** catches errors early
  - Cannot add *int* key to *string* hashtable

- **Generics**: in Java, C#, VB (borrowed from ML)
Type Inference

How DOES Ocaml figure out all the types ?!
Polymorphic Types

• Polymorphic types are tricky

• Not always obvious from staring at code

• How to ensure correctness?

• Types (almost) never entered w/ program!
Polymorphic Type Inference

• Computing the types of all expressions
  - At compile time: statically Typed

• Each binding is processed in order
  - Types are computed for each binding
  - For expression and variable bound to
  - Types used for subsequent bindings

• Unlike values (determined at run-time)
Polymorphic Type Inference

• Every expression accepted by ML must have a valid inferred type

• Can have no idea what a function does, but still know its exact type

• A function may never (or sometimes terminate), but will still have a valid type
Example 1

```ocaml
let x = 2 + 3;;
let y = string_of_int x;;
```
Example 2

```
let x = 2 + 3;;
let y = string_of_int x;;
let inc y = x + y;;
```
Example 5

```ocaml
let rec cat xs =
    match xs with
    | [] -> ""
    | x::xs -> x^(cat xs)
```
ML doesn’t know what function does, or even that it finishes only its type!

```
let rec cat xs =
  match xs with
  | []     -> ""
  | x::xs  -> x^(cat xs)
```

```
let rec cat xs =
  match xs with
  | []     -> cat []
  | x::xs  -> x^(cat xs)
```
Example 5

```ocaml
let rec map f xs =
  match xs with
  | []      -> []
  | x::xs'  -> (f x)::(map f xs')
```

Thursday, November 1, 2012
Example 5

let rec map f xs =
    match xs with
    | []     -> []
    | x :: xs' -> (f x) :: (map f xs')

“Generalize” Unconstrained Vars

(‘a -> ‘b) -> ‘a list -> ‘b list
Example 7

```ocaml
let rec fold f cur xs =
  match xs with
  | []     -> cur
  | x::xs' -> fold f (f cur x) xs'
```
Example 11

```ocaml
let fool f g x =
  if f x
  then x
  else g x
```
Example 12

```ocaml
let foo2 f g x =
  if f x
  then x
  else foo2 f g (g x)
```